Infrastructure de télédomotique

Travail de diplôme

Auteur : Mathieu Despont
Professeur : Markus Jaton
Filière : Télécommunication
Date : 18 décembre 2003

Table des matières

1 Motivation

Le terme *télédomotique* désigne une domotique à distance. Donc, le fait de pouvoir piloter à distance une *maison intelligente*.

Depuis toujours le fait de pouvoir commander à distance les appareils dans sa maison fait rêver les gens. On trouve une foule d'exemples de télédomotique dans les ouvrages de science-fiction. Mais avec l'évolution actuelle de la technologie, la télédomotique sort du domaine de la science-fiction, et devient réalisable.

La télédomotique est particulièrement intéressante pour surveiller une maison.

Imaginez que vous partez en vacances, vous avez pensé à tout, sauf à un petit détail : *Qui, va donner à manger à votre poisson rouge durant votre absence ?* En effet, par une incroyable malchance, la voisine qui s'acquittait habituellement de cette tâche est également absente !

Songeant à l'absence simultanée de votre voisine et de vous même, vous vous dites également, que le voisinage va être bien vide... Une aubaine pour les cambrioleurs !

Devez-vous donc renoncer à vos vacances ?
Non ! Grâce à une installation de télédomotique, vous pourrez nourrir votre poisson rouge depuis votre lieu de vacances. Vous pourrez également surveiller votre maison au travers de caméras et éloigner les cambrioleurs en faisant croire à une présence en allumant et éteignant les lampes.

2 Quelques exemples de services

L'exemple ci-dessus démontre l'utilité d'une installation de télédomotique dans votre propre maison. On peut également imaginer qu'une telle installation soit disponible dans votre chalet perdu dans la montagne.
Dans ce cas-ci, à la place de donner à manger à votre poisson rouge, vous préférez certainement allumer le chauffage quelques heures avant votre arrivée, afin de bénéficier d'une température agréable lorsque vous entrerez dans votre chalet.

Une fois confortablement assis dans votre fauteuil au coin du feu dans votre chalet, vous apercevez dans le programme TV que votre émission préférée passe le soir même à la télévision. Comble de malchance, vous avez toujours refusé d'installer la télévision dans votre chalet. Ce soir, vous regrettez votre choix !
Mais tout va s'arranger, grâce à votre installation de télédomotique, vous allez pouvoir programmer depuis votre chalet l'enregistrement de votre émission de TV favorite sur le magnétoscope de votre maison. Vous pourrez ainsi regarder l'émission une fois de retour chez vous.

Une installation de télédomotique peut également être un précieux outil de travail pour le gardiennage d'un refuge de montagne.
Il devient ainsi plus aisé de vérifier, depuis la vallée, l'état des réserves du refuge, de communiquer avec les éventuels occupants et d'obtenir des informations météorologiques sur l'environnement du refuge.

On peut encore imaginer une foule d'exemples, dans lesquels peut intervenir une infrastructure de télédomotique.

3 Objectifs de ce projet

Il existe donc, dans une maison, une foule d'appareils ou de fonctions qu'il serait intéressant de pouvoir commander à distance. Par la suite, tous ces appareils ou fonctions seront désignés sous le terme de *service*.
Un *service* est une entité avec laquelle ont peut interagir. Un service peut posséder des états et il propose une ou plusieurs fonctions.
Dans un contexte purement domotique, ces fonctions sont celles de l'appareil que le service représente. Par exemple, une machine à café possède la fonction *Chauffer l'eau* et la fonction *Verser le café*. De plus il est possible de se renseigner sur l'état de la réserve d'eau.
Quant à lui, le service *lampe* dispose des fonctions *Allumer*, *éteindre* et d'une fonction nous indiquant l'état de la lampe.

En observant ces deux exemples, on remarque, qu'un *service* peut se présenter sous de multiples formes.

De plus, il existe une multitude de moyens de communication entre le *service* et le *client* qui désire utiliser le service.

Le but du présent projet, comme décrit dans le cahier des charges [1], est donc de concevoir une infrastructure qui permette à un *client* d'utiliser un *service* quelconque sans qu'il ait à se soucier du moyen de communication employé.

Le client opère toujours d'une manière semblable pour piloter une machine à café ou une lampe.
L'infrastructure se charge de trouver le moyen de communication adéquat.

Du point de vue d'un fournisseur de services, l'infrastructure de télédomotique est très intéressante. Le constructeur de chauffage qui veux mettre à disposition un service *chauffage*, doit seulement rendre compatible son service avec l'infrastructure de télédomotique. Il n'a plus besoin de se soucier de la manière avec laquelle le client va accéder à son service.

4 Service de découverte et d'utilisation de service

Comme décrit précédemment, le rôle d'une infrastructure de télédomotique est d'offrir un accès, via un moyen de communication quelconque, à des services quelconques.
Malgré cette grande diversité, l'infrastructure doit être capable de communiquer avec tous les services.
Pour assurer cette communication, on peut se poser quelques questions :
- Comment représente t-on un service ?
- Comment l'infrastructure connaît-elle quel moyen de communication utiliser avec quel service ?
- Une fois le moyen de communication trouvé, comment connaître l'adresse du service ?

La solution se trouve dans la définition d'une interface commune présente entre chaque service et l'infrastructure. Cette interface joue le rôle du morceau de *colle* entre le service et l'infrastructure.

Le problème n'étant pas totalement nouveau, il existe ce que l'on appelle des SDUS, des *Services de Découverte et d'Utilisation de Services.*

Un SDUS fournit les moyens pour décrire un service, ainsi que pour annoncer sa présence et sa localisation.

Différents SDUS sont disponibles sur le marché, entre autre : UPnP, Salutation et Jini.

4.1 UPnP

UPnP *Universal Plug and Play* [2] est un SDUS défini par une multitude d'entreprises réunies sur le forum UPnP. Ces entreprises sont fortement influencées par Microsoft semble-t-il.
Une architecture UPnP est composée de plusieurs *dispositifs* qui sont des conteneurs logiciels de services. Un *dispositif* représente un appareil, ou plusieurs s'il est le dispositif racine. Par exemple, un magnétoscope peut être composé d'un service de *tuner* et d'un service d'horloge.
Un *dispositif* détient une adresse IP et peut être piloté par un *point de contrôle* via une liste *d'actions* pour modifier ses *variables d'état*.

Dans une architecture UPnP, les services ainsi que leurs caractéristiques sont décrits au format XML *eXtensible Markup Language* dans une liste que détient le *dispositif*.

Les communications entre les services se font via les protocoles SOAP *Simple Object Access Protocol*, Gena *Generic Event Notification Architecture* et SSDP *Simple Service Discovery Protocol* qui tous trois utilisent HTTP pour véhiculer leurs messages.

4.2 Salutation

Salutation [3] se veut être un SDUS complètement indépendant de l'influence de toute entreprise. Cette architecture se veut indépendante de tout processeur, OS, langage et système de communication. C'est une architecture complètement libre et gratuite.

Ce SDUS fournit un certain formalisme pour décrire des services. Son principe de fonctionnement est basé sur la recherche du service demandé et sur le téléchargement de son *driver* afin que le client puisse l'utiliser.
Salutation tend à rendre l'installation de *drivers* totalement transparente. Salutation vise particulièrement le monde des imprimantes et des FAX.

4.3 Jini

JINI [4] est la solution proposée par SUN au problème du *Plug and Play* réseau pour les services. Une architecture JINI est utilisée par des clients pour découvrir et utiliser des services disponibles sur un réseau informatique. Une architecture JINI est donc le *Génie* du réseau informatique capable d'orienter les utilisateurs sur les différents services disponibles.

JINI à la particularité de reposer sur le langage java[1]. En effet, le langage java présente déjà intrinsèquement des mécanismes qui satisfont les besoins d'un SDUS. La description d'un service avec JINI se fait avec le mécanisme d'*interface* intrinsèque au langage. (Une interface est la description des méthodes d'une classe sans son implémentation.)

La communication entre les différents services se fait par RMI qui est le mécanisme de RPC propre à Java. RMI à l'avantage d'être un protocole très léger par rapport à d'autres protocoles du type de SOAP qui sont utilisés par un SDUS tel que UPnP.
En effet, si le protocole utilisé ne semble pas être de la plus grande importance dans une utilisation sur un réseau haut-débit, dans le cas d'un refuge de montagne ne disposant, comme réseau d'accès, que de GPRS, l'avantage d'un protocole léger se remarque, tant du coté des temps d'attente, que du coté du porte-monnaie.

[1] Les spécifications sont indépendantes du langage, mais la seule implémentation disponible est en Java

La cohérence des mécanismes utilisés par Jini permet de faire ”voyager” des objets entre différents clients ou services, sur différentes machines virtuelles java *JVM*. Cette technique est un avantage pour créer une véritable application répartie faisant vraiment abstraction du réseau.

Un autre avantage de Java est son aspect multi-plateforme qui permet de s'affranchir du matériel sur lequel vont fonctionner les services. Matériel qui dans le cas de la domotique peut être très hétéroclite. La nouvelle version Jini 2.0 supporte également SSL et Kerberos ce qui fait de Jini un SDUS très sûre.

Pour toutes ces raisons, Jini parait être le SDUS qui est le mieux disposé à la construction d'une infrastructure de télédomotique.

5 Jini

5.1 Principe de base

L'architecture Jini est basée sur ce que l'on appelle une *fédération de services.*
Dans une fédération de services, il y a :
– des services ;
– des clients ;
– un service de recherche.

Toute l'architecture Jini repose sur le principe du *code mobile*.
A la question : Comment un client fait il pour utiliser un service qu'il ne connaît pas ?
On peut répondre par une autre question : Comment fait un ordinateur pour utiliser une carte qu'il ne connaît pas ?
Et bien, pour qu'un ordinateur puisse utiliser une carte inconnue, on lui fourni le *driver* adéquat. Chaque carte graphique a son *driver* et chaque imprimante est fournie avec son pilote d'impression.
Dans l'architecture Jini, c'est pareil. Chaque service possède un *driver* et chaque client qui désire utiliser un service contacte ce dernier via son *driver*.
Dans la terminologie Jini, un *driver* s'appelle un *proxy*.

Si l'on reprend l'exemple de l'installation d'une imprimante inconnue. En général, l'installation du pilote doit être faite par l'utilisateur. Ce dernier doit, pour commencer, posséder le pilote. Ici, en général deux choix se présentent à l'utilisateur, soit il entreprend une fouille archéologique dans le carton de l'imprimante pour retrouver le CD sur lequel est censé se trouver le pilote, soit il va directement télécharger le pilote sur le site web du constructeur de l'imprimante.

La seconde solution est la plus proche de ce qui se passe dans une architecture Jini. En effet, les *proxies* sont également téléchargés depuis un serveur web.
Pour télécharger le pilote d'une imprimante, logiquement, il suffit de se diriger sur le site web du constructeur. En revanche, où faut-il télécharger le *proxy* d'un service Jini ?
C'est là qu'intervient le véritable *génie* de l'architecture : *le service de recherche*.

5.2 Service de recherche

Le service de recherche est à l'architecture Jini, ce que Google est au web.
Google (ou n'importe quel autre moteur de recherche) permet à un utilisateur web de trouver le site qu'il désire. Le service de recherche Jini, est l'entité qui permet à un client Jini de trouver et

télécharger le *proxy* d'un service.
Un service de recherche stocke tous les *proxies* d'une fédération de service.

Le service de recherche est donc le *moteur de recherche* de la fédération de service. Il est utilisé pour trouver tous les services. Mais comment fait un client pour trouver le service de recherche ?
C'est une question pertinente, comment trouver le service qui permet de trouver les services ?

Il existe deux solutions :
– Soit, le client connaît l'adresse du service de recherche.
– Soit, le client effectue une recherche dans tout son voisinage.
La première solution est identique a celle que l'on utilise pour les moteurs de recherche sur le web, il faut connaître l'adresse du service de recherche à l'avance. Pour ce faire, il existe plusieurs canaux, la presse, la publicité, le bouche à oreille, etc...

La deuxième solution est plus élégante. Elle consiste à diffuser des requêtes sur une adresse *multicast* correspondant au groupe des services de recherche. Lorsqu'un service de recherche reçoit la requête d'un client il lui envoie son adresse. Cette méthode permet de découvrir tous les services de recherche disponibles à portée de diffusion *multicast*. Cette méthode est plus élégante, mais elle est également limitée par la portée de la diffusion *multicast*.

Il n'y a pas seulement les clients qui ont besoin de connaître l'adresse d'un service de recherche. Les services ont aussi besoin de cette information pour exporter leur *proxy*. Ils utilise pour ce faire également les mêmes techniques.
Le service de recherche fourni avec les paquetages de base de Jini s'appelle *Reggie*.

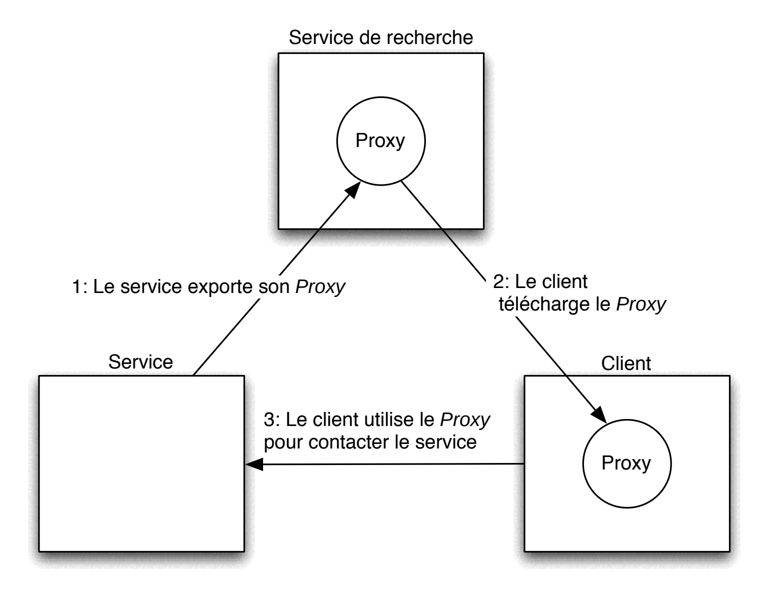


FIG. 1 – Cheminement du *proxy*

5.3 Hébergement du *proxy*

Chaque service dispose d'un *proxy* qui est mis à disposition sur un serveur web. Ce *proxy* est exporté sur le service de recherche. Un client qui désire utiliser un service va interroger le service de recherche pour télécharger le *proxy* voulu. Ce dernier étant originaire du service, il connaît le moyen de commu-

nication et l'adresse à employer pour communiquer avec le service. Ce *proxy* est donc un morceau de code mobile provenant du service et qui va être exécuté dans l'application cliente.

Il revient donc au concepteur du service de créer et de mettre à disposition, du service de recherche, un *proxy*. Les *proxies* sont mis à disposition sur un serveur web. Généralement, on dédie un serveur web sur un port particulier du serveur.

Cependant, on peut très bien utiliser n'importe quel serveur web pour héberger les *proxies*. On peut même utiliser un serveur web qui n'est pas du tout sur la même machine, ou à la même adresse que le service.
Dans ce cas, la seule contrainte est de disposer d'un programme qui se charge, à la place du service, de signaler, au service de recherche, la présence et la localisation d'un nouveau *proxy*.

Cette manière de séparer entièrement les diverses fonctions d'un service est la solution proposée pour utiliser un service limité en ressource et en mémoire, ou ne disposant pas d'un réseau lui permettant d'exporter son *proxy*.
En effet, il n'est pas rare de voir en domotique des appareils qui ne disposent pas de processeur véloce, d'une mémoire d'éléphant et surtout d'une machine virtuelle Java !
Cette solution pour utiliser de tels appareils dans une architecture jini est donc indispensable pour concevoir une infrastructure destinée à accueil des applications de domotique.
Le principe détaillé de cette solution est décrite dans le projet *Surrogate* [7] de la communauté Jini.

5.4 Moyen de communication entre un client et un service

Couramment dans une fédération de services Jini, le *proxy* communique avec son service via des appels de méthodes à distance RMI.

Ce moyen de communication permet à des applications hébergées à des endroits différents, sur des machines virtuelles Java différentes, de communiquer entre elles comme si elles étaient sur la même machine. RMI est le moyen de communication qui est le plus transparent au niveau du réseau.

Jini enrichi même RMI.
Par défaut, RMI utilise le protocole JRMP, *Java Remote Method Protocol*, construit sur TCP. Par la suite, afin de supporter d'autres types de réseaux, d'autres RMI ont été développés. Il y a par exemple, RMI sur IIOP (le protocole de CORBA), RMI sur HTTP, RMI sur SSL et même RMI sur FireWire.
Toutefois, cette diversité de RMI pose un problème. Chaque type de RMI a sa propre manière d'être programmé. Il faut donc concevoir à l'origine une application en fonction du type de RMI qu'elle va employer.

Afin de pallier à ce problème, Jini fournit JERI, *Jini Extensible Remote Invocation*. C'est une couche intermédiaire qui permet d'écrire une application RMI sans se préoccuper du protocole sous-jacent. Celui-ci peut être choisi à l'exécution. Le type de protocole que l'application utilise est décrit dans un fichier de configuration Java. L'application lit le fichier à chaque démarrage et construit les bons objets de communication. Cette technique permet de changer de type de RMI sans avoir à recompiler l'application. JERI s'occupe également de compiler un bout serveur et un bout client à la volée. L'utilisation du compilateur *rmic* n'est donc plus requise.

Cependant, dans certains cas, RMI n'est pas utilisable. En effet, certaines applications ne sont pas capables d'utiliser RMI comme moyen de communication, ou encore le réseau sous-jacent reliant l'application et le client ne supporte pas RMI, et enfin, l'utilisation de RMI n'est peut être pas voulue par les concepteurs du service.

Dans tous ces cas, il suffit d'inclure dans le *proxy*, le code nécessaire à l'utilisation par le client du moyen de communication voulu, permettant de contacter le service.

Par exemple, un service de web cam propose des images. Transmettre des images par RMI n'est pas chose des plus aisée et des plus adéquate. Il est donc nettement plus simple d'inclure, dans le *proxy* du service de web cam, l'adresse permettant de récupérer l'image par le protocole HTTP. Car c'est, comme le nom de *Web* cam l'indique, le protocole le plus approprié à cette tâche. Dans ce cas, l'architecture jini a pour seul rôle de fournir à un client une adresse.

5.5 L'objet `ServiceItem`

Plus haut, nous avons parlé du *proxy* qui est le bout de code mobile utilisé par le client pour communiquer avec le service. Nous avons également souligné que les clients obtiennent le *proxy* voulu en interrogeant le service de recherche.

Toutefois, comment fait le client pour choisir le *proxy* adéquat parmi les *proxies* de tous les services de la fédération ?
Et bien, le *proxy* n'est pas le seul objet exporté au service de recherche. Le *proxy* est tout d'abord encapsulé dans l'objet `ServiceItem`. Cette objet `ServiceItem` contient trois objets :
- L'objet `service` qui contient le *proxy*.
- L'objet `serviceID` qui contient l'identificateur unique du service.
- L'objet `attributeSets` qui peut contenir toutes sortes d'attributs.

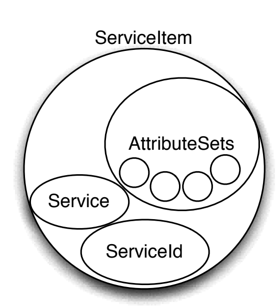


FIG. 2 – L'objet `ServiceItem` contient toutes les informations utiles pour sélectionner le service.

Plus précisément le `serviceID` est un nombre de 128 bits généré selon un algorithme qui permet de garantir son unicité. Ce nombre sert d'identificateur pour les services. Un `serviceID` peut être attribué de manière fixe à un service au moment de son écriture, ou plutôt, généré dynamiquement au moment de son enregistrement dans le service de recherche.

L'objet `attributeSets` est un tableau d'objets de type `Entry`. Il peut contenir toutes sortes d'attributs permettant de décrire le service.

5.6 Entry

Les attributs contenus dans le tableau `attributeSets` sont de type `Entry`. Ce type est en faite une interface. N'importe quel objet peut être utilisé comme attribut du service et être placé dans le tableau `attributeSets` pour autant qu'il implémente l'interface `Entry`.

Il existe tout de même une restriction, le type `Entry`, a été créé pour signaler, non seulement la fonction de l'objet, mais aussi que cet objet doit être sérialisable. Donc, il n'est pas possible de définir comme `Entry` un type primitif (float, int, etc...).

Le paquetage `net.jini.lookup.entry` définit un certain nombre d'objets de type `Entry` qui peuvent être utilisés pour indiquer la localisation physique d'un service.
Entre autres, les objets, `Address`, `Location`, `Name`, et `Comment` sont les plus utiles dans le cadre de la conception d'une application de domotique.
Il est ainsi possible, pour un client, de sélectionner, dans un service de recherche, tous les services se trouvant dans une telle chambre de la maison située à une certaine adresse postale.

5.7 Autres services fournis avec l'architecture Jini

L'architecture Jini est complexe. Pour l'obtenir, il faut télécharger le Starter Kit (JSK) sur le site officiel des la communauté Jini [4]. Le JSK contient la documentation, un exemple, ainsi qu'une foule d'archives JAR contenant toutes les classes nécessaires à la conception d'applications basées sur Jini. En plus de ceci, le JSK contient quelques services qui peuvent être utiles à la mise en place d'une fédération de service.
Il y a entre autres :

- Un serveur web minimal destiné délivrer les *proxies* dans des archives JAR.
- *Reggie*, un service de recherche.
- *Mahalo*, un manager de transaction entre les services.
- *Norm*, un gestionnaire de bail pour les *proxies*.
- *Mercury*, une boîte aux lettres d'événements.
- *Outrigger*, un gestionnaire JavaSpaces(TM), un système de répertoires répartis pour stocker des objets. Ce système permet de partager des données entre plusieurs applications.

6 Construction d'un service

6.1 Version

Lorsque l'on désire développer des applications avec Jini, la première chose à faire est de télécharger sur le site de la communauté Jini [4], le JSK, *Jini Starter Kit*. Actuellement, depuis juin 2003, c'est la version 2.0 qui est disponible. Cette version apporte nombre de nouveautés et de changements par rapport aux versions précédentes. Les applications décrites dans ce document ont toutes été écrites avec la version 2.0 de Jini.

6.2 Environnements de programmation

En développant des applications avec une architecture telle que Jini, on est très vite perdu dans la masse de fichiers sources, fichiers de configuration, archives JAR et scripts en tous genres.
Il est donc important de bien organiser la gestion de tous ces fichiers. Au moment de la compilation d'une application, nombre de fichiers sont pris en compte et quelques archives JAR doivent être créées.
Il est donc très utile de disposer d'un environnement de programmation qui permet d'automatiser certaines tâches.

Dans le JSK, c'est l'outil make qui est proposé, plusieurs *MakeFile* sont disponibles pour recompiler toutes les classes de l'architecture.

Toutefois, via le projet *Davis* [8] de la communauté jini, il existe un script ANT [9] qui lui aussi permet de recompiler toutes les classes.

Depuis l'été 2003, est arrivé sur le marché, le premier IDE entièrement dédié à la programmation et la gestion d'applications Jini. Il s'agit de IncaX [10]. Cet environnement de programmation existe en trois versions, une version *community* qui est gratuite mais limitée, une version *Entreprise*, qui permet en plus la gestion d'une fédération de service, et une version intermédiaire disponible pour une modique somme.

IncaX comporte plusieurs *wizards* qui permettent de créer et d'exécuter un service en répondant simplement à des questions. C'est remarquablement efficace. Malheureusement, les *wizards* utilisent des classes propriétaires pour créer les services. Dans un soucis de compatibilité globale, malgré que l'on puisse également utiliser IncaX sans les *wizards* , donc sans les classes propriétaires, cet environnement de programmation n'a pas été retenu pour le présent projet.

Entre le puissant mais complexe outil make et son équivalent à la syntaxe plus verbeuse ANT, le choix c'est porté sur ANT, principalement pour des raisons de simplicité d'emploi.

6.3 Etapes de conception

L'écriture du code d'un service se fait en trois étapes :

1. Ecriture de l'interface du service.
2. Ecriture du *proxy* du service.
3. Ecriture de l'implémentation du service.

Premièrement, on va décrire dans une interface les méthodes qui caractérisent le service. Ensuite, on écrira le code du *proxy*, donc le code que le client va utiliser pour communiquer avec le service. Et enfin, on écrira l'implémentation du service, sa fonctionnalité.

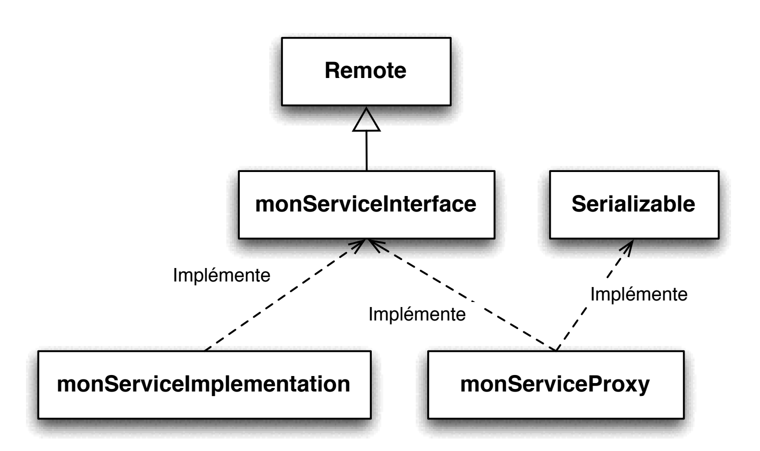


FIG. 3 – Organisation des classes composant le service

6.3.1 Ecriture de l'interface du service

L'exemple suivant va être basé sur un des services qui a été réalisé dans le cadre de ce projet pour montrer les possibilités de l'infrastructure de télédomotique.

Il s'agit d'un service qui pilote un distributeur de nourriture pour poissons.
Le service comporte deux fonctions, une servant à délivrer aux poissons une certaine quantité de nourriture, une autre permettant d'interroger le distributeur pour connaître la quantité de nourriture restante.



FIG. 4 – Un distributeur de nourriture pour poissons

```
package foodDistributor;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface foodDistributor extends Remote {
         boolean giveFood(Integer quantity ) throws RemoteException;
         Integer remainingQuantity() throws RemoteException;
}
```

6.3.2 Ecriture du *proxy*

Le *proxy* implémente l'interface `Serializable` et l'interface `foodDistributor` du service.
Le *proxy* est créé par le service. Ce dernier au moment de la construction passe en paramètre le bout client utilisé pour effectuer des appels de procédure à distance.

```
// Constructeur du proxy
foodDistributorProxy(foodDistributor serverProxy){
        this.serverProxy = serverProxy;
}
```

Ensuite, il est temps d'implémenter les méthodes de l'interface du service. Le bout client ici appelé `serverProxy` est utilisé pour pour contacter le serveur via RMI.

```
public boolean giveFood(Integer quantity ) throws RemoteException {
        return serverProxy.giveFood(quantity);
}
public Integer remainingQuantity() throws RemoteException {
        return serverProxy.remainingQuantity();
}
```

6.3.3 Ecriture de l'implémentation du service

Le but principal de l'implémentation du service est, comme son nom l'indique d'implémenter le service, mais pas seulement.

Voici brièvement les différentes étapes que le service va accomplir pour être disponible :

1. La méthode `main` crée un nouveau service. Dans notre cas `foodDistributorImpl`.
2. Créer un `exporter` du type RMI voulu.
3. L'`exporter` est utilisé pour créer les bouts client et serveur. Le bout server est exporté.
4. Le *proxy* du service est créé. On lui passe en paramètre le bout client.
5. Un objet `LookupDiscovery` est créé. Il sert à trouver le service de recherche.
6. Un `JoinManager` est créé. Il se charge de joindre le service à la fédération de services.
7. Il ne faut pas oublier d'implémenter les fonctionnalités du service.

Le rôle de l'`exporter` est de représenter le moyen de communication sous-jacent. La configuration la plus classique est la construction d'un `exporter` pour RMI sur TCP.
Pour construire un `BasicJeriExporter` il faut lui fournir un point d'écoute et une "fabrique" de qui construit les bouts client et serveur.
Dans le cas de TCP, le point d'écoute est un port TCP.

```
exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                          new BasicILFactory());
```

Le service crée un *proxy* en lui fournissant le bout client qui lui permettra de contacter le service.

```
foodDistributorProxy smartProxy = new foodDistributorProxy(serverProxy);
```

Ensuite, on crée un `LookupDiscovery`. Ce dernier va se charger de découvrir un service de recherche en effectuant des diffusions *multicast*. On peut préciser un groupe, ici le groupe *teledomotique*. En effet, il est possible, dans une même fédération de service, de grouper des services.

Un service de recherche peut ainsi être exclusivement réservé au stockage des services appartenant à un certain groupe.

```
DiscoveryManagement discoveryManager = new LookupDiscovery(
                 new String[] { "teledomotique" }, config);
```

L'étape suivante consiste à créer un `JoinManager`.
Ce gestionnaire va s'occuper de créer le `serviceItem`, d'y encapsuler le service ainsi que ses attributs, et enfin, il va signaler au service de recherche qu'un nouveau *proxy* est disponible sur un serveur web à une certaine adresse.

Le `JoinManager` est donc le gestionnaire qui permet à un service de joindre la fédération de services. Il est responsable de toutes les communications avec le service de recherche.
Dans le code ci-dessous, on inclus dans les attributs le nom du service ainsi qui son adresse postale et sa localisation plus précise.

Parmi tout les gestionnaires que l'on indique au gestionnaire `JoinManager`, on a omis de préciser un `leaseManager`, on utilise ainsi la durée de bail par défaut du service de recherche *Reggie* qui est de 5 minutes. Toutes les cinq minutes, le service de recherche va donc re-télécharger le *proxy* du service. Ce mécanisme de bail permet de maintenir le service de recherche à jour. Il est ainsi, à cinq minutes près, toujours au courant de la dernière modification du service, ou de sa dernière localisation.

```
JoinManager joinManager = new JoinManager(
     smartProxy, new Entry[] { name, address, location},
     getServiceID(), discoveryManager, null /* leaseMgr */, config);
```

La dernière étape consiste à implémenter l'interface du service. Dans notre cas, il s'agit de l'interface `foodDistributor`.
Le service n'étant qu'une démonstration, il ne pilote pas vraiment de distributeur de nourriture. Il se contente d'écrire quelques ligne dans une console.

6.4 Configuration

Pour chaque service, il est bon de pouvoir obtenir quelques informations sur lui. Ainsi, dans les attributs sont placés, le nom du service, l'adresse postale de l'endroit où il se trouve et sa localisation précise (3ème étage chambre de gauche).

Par exemple, une personne qui utilise chez elle un service de distributeur de nourriture pour ses poissons, a indiqué dans son service que c'est le service pour nourrir les poissons rouges de l'aquarium du salon. Si un jour il décide de mettre le distributeur sur l'aquarium des piranhas de la cuisine, il est obliger de re-compiler le service !

De même, un fabricant de distributeur de nourriture pour poissons doit compiler différemment le service pour chaque client !

Il y a, dans chaque service, des informations, ou des objets, qu'il est bon de pouvoir modifier au démarrage sans devoir re-compiler entièrement le service.

Pour résoudre ce problème, on peut utiliser un fichier de configuration.

Un fichier de `configuration` à la Java utilise une syntaxe très proche de celle du langage Java. De plus, des utilitaires sont fournis dans le paquetage *com.sun.jini.config*.

Un de ces utilitaires, `getHostName()`, permet d'obtenir le nom de l'hôte sur lequel est démarrée l'application.

Une application peut ainsi, au moment de son exécution, aller chercher un objet particulier dans un fichier de configuration.

Un programmeur utilisera la méthode suivant pour obtenir un objet d'un fichier de configuration :
`Object Configuration.getEntry(String component, String name, Class type)`

Pour utiliser un fichier de configuration, généralement, on passe le nom du fichier de configuration en argument à la méthode `main` de l'application.

L'utilisation d'un fichier de configuration est très utile pour choisir un `exporter`. Ainsi, peu importe le type de RMI utilisé finalement, l'application est écrite de la même manière. Seul le fichier de configuration change. Il est ainsi possible de choisir le type de RMI à utiliser juste en choisissant parmi un des `exporter` suivant :

```
 JeriExporterDemo {exporter = new BascJeriExporter(
                                  TcpServerEndpoint.getInstance(0),
                                  new BasicIlfactory);}
 JrmpExporterDdemo{ exporter = new JrmpExporter();}
 IiopExporterDdemo{ exporter = new IiopExporter();}
```

Dans cette même optique, le gestionnaire chargé de découvrir le service de recherche est aussi créer à partir d'un fichier de configuration. Cela permet de changer facilement le type de gestionnaire, ainsi que le groupe auquel il est affilié.

Bien entendu, les informations spécifiques à la localisation physique du service sont également décrites dans un fichier de configuration. Ainsi la configuration d'un service de WebCam situé dans la salle B08 à l'étage B de l'eivd ressemble à ceci :

```
WebCam{
  name = new Name("Web cam");
  address = new Address("Rte de Cheseaux","eivd"/*organization*/,
           null/*Unit*/, "Yverdon",null/*state*/,"1401","Suisse");
  location = new Location("B", "B08", null/*Building*/);
}
```

6.5 Compilation

La compilation du service est assez complexe. Pour simplifier la tâche, celle-ci va être exécutée par un script ANT.

Cependant, avant toute chose, il faut organiser les fichiers de manière claire.



FIG. 5 – Arborescence des fichiers d'un service

Premièrement, on va créer un dossier de base dans lequel sont placés tous les services, ainsi que le dossier web qui contient toutes les archives JAR encapsulant les *proxies* des services.

Ensuite, pour chaque service, on défini un dossier de base depuis lequel sont lancés tous les scripts concernant le service. Ce dossier contient plusieurs sous-dossiers :
– classes/, contenant toutes les classes du service et du client associé.
– config/, contient tout les fichiers de configuration relatifs au service.
– lib/, contient les classes du service groupées dans une archive JAR.
– scripts/, contient les scripts pour démarrer le service.
– src/, contient les fichiers source du service.

Dans ce même dossier de base du service, on trouve deux fichiers : build.xml et build.properties. Ce sont, respectivement, le fichier Ant de compilation et son fichier de configuration.

Plusieurs tâches Ant sont utilisées pour effectuer la construction du service. Il y a les tâches :
– service.compile, permettant de compiler toutes les classes concernant le service.
– client.compile, permettant de compiler toutes les classes concernant le client.
– jar, qui compile toutes les classes et les groupe dans les archives JAR adéquates.

Il y a plusieurs archives JAR qui sont créées. Il y a foodDistributor-service.jar et foodDistributor-client.jar qui contiennent toutes les classes du service ou du client. Il y a également foodDistributor-service-dl.jar qui contient le *proxy* et l'interface du service, mais pas l'implémentation. Cette archive est placée dans le dossier web qui est accessible sur un serveur web.

Le fichier de configuration build.properties contient quelques informations utiles pour la compilation. Il y a, entre autres, les noms des dossiers où seront pris ou placés les fichiers utilisés ou créer par Ant. De plus, c'est également dans ce fichier que l'on indique le chemin d'accès à la bibliothèque des archives JAR de Jini. En effet, il ne faut pas oublier d'installer la bibliothèque pour que le service puisse aller chercher les classes dont il a besoin.

Pour effectuer une compilation complète et construire les archives JAR. Il faut exécuter, dans un terminal, depuis le dossier du service, la commande suivante :

```
[hibou:foodDistributor] mdespont% ant jar
```

Cette commande exécute avec Ant la tâche jar.

Petite précision, le fichier build.xml fourni, contient des tâches qui font la compilation d'un service disposant d'une interface utilisateur graphique. Cette notion sera traitée plus tard (section : 9.2).
Si l'on veut compiler un service simple sans interface graphique, il faut donc modifier le fichier build.xml pour lui enlever tous ce qui concerne celle-ci.

7 Construction d'un client

Ci-dessus, nous avons brièvement parlé d'applications clientes du service. La section présente va décrire les principes de base pour la création d'une telle application cliente capable d'utiliser un service.

7.1 Etapes de conception

Un client est beaucoup plus simple à concevoir qu'un service. Un client passe par trois étapes pour utiliser un service :

1. Trouver un service de recherche.
2. Définir des critères de recherche.
3. Obtenir du service de recherche, les services correspondant aux critères.

Pour trouver un service de recherche, le client utilise un gestionnaire : le `ServiceDiscoveryManager`. Ce gestionnaire est généralement tiré d'un fichier de configuration, et ceci dans le but de pouvoir changer facilement le groupe au quel appartient le service de recherche désiré.

La seconde étape consiste à définir les critères de recherche qui permettront de trouver le service adéquat. Les critères sont définis par l'intermédiaire d'un objet `ServiceTemplate`. Cet objet est le "petit frère" du `ServiceItem` dont nous avons parlé précédemment (section : 5.5).

Le `ServiceTemplate` reprend les mêmes champs qu'un `ServiceItem`, à la différence que le *proxy* est remplacé par le nom de l'interface du service.

Cette ressemblance entre les objets n'est pas un hasard.

En effet, le but du client est d'obtenir le `ServiceItem` du service voulu. Pour ce faire, le client appel la méthode `lookup` du gestionnaire `ServiceDiscoveryManager` et passe en paramètre le `ServiceTemplate` qui vient d'être construit. En retour, la méthode envoie un tableau contenant tous les `ServiceItem` correspondant aux critères de recherche.

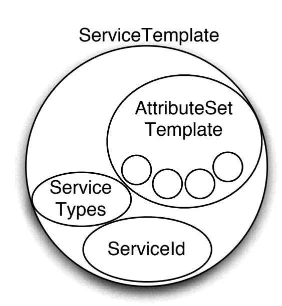


FIG. 6 – Composition de l'objet `ServiceTemplate`

Le `ServiceTemplate` n'est pas le seul paramètre passé à la méthode `lookup`. On peut également fournir à cette méthode un filtre qui permet d'affiner la recherche une fois les `ServiceItem` téléchargés. De plus, la durée maximale de la recherche est également fournie.

```
ServiceItem serviceItem = serviceDiscovery.lookup(
    serviceTemplate, null /*ServiceItemFilter */,
    Long.MAX_VALUE /*Search duration*/);
```

Dans le cas ci-dessous, la recherche va se faire uniquement sur le nom de l'interface du service.
Un objet `null` n'est pas pris en compte pour la recherche, il s'agit du joker.

```
new ServiceTemplate(null new Class[] {foodDistributor.class},null)
```

Ainsi, un client désirant obtenir tous les services disponibles dans la fédération effectuera une recherche à l'aide d'un `ServiceTemplate` dont tous les champs sont à `null`.

7.2 Compilation

La compilation d'un client s'effectue de la même manière que celle du service. La seule différence, réside dans le fait que le client n'a pas besoin de mettre une partie de son code à disposition sur un serveur web.

8 Mise en service de la fédération de services

Une fédération de service est composée, de services, de clients et d'un service de recherche. Dans cette section nous allons décrire comment mettre en place tout ceci.

8.1 Démarrage du service de recherche

Le service de recherche est le service le plus important d'une fédération. Nous allons utiliser ici, le service de recherche fourni avec le *Starter kit*. Il s'appelle *Reggie*.

Le nombre de paramètres à indiquer pour démarrer *Reggie* étant assez important, toutes les instructions ont été placées dans un *shell script* nommé *start-reggie.sh* et placé dans le dossier des scripts du service.

Nous allons démarrer *Reggie* via l'archive *start.jar* qui se trouve dans la bibliothèque des archives de base de Jini.

```
JINIHOME=/Users/mdespont/Documents/Ecole/teledomotique/jini2_0

java -Djava.security.manager=                        \
     -Djava.security.policy=config/policy.all        \
     -Djskhome=$JINIHOME                             \
     -jar $JINIHOME/lib/start.jar                    \
     config/start-jrmp-reggie.config
```

Ce script doit être lancé depuis le dossier de base du service.

```
[hibou:foodDistributor] mdespont% sh scripts/start-reggie.sh
```

Nous démarrons ici, une instance de *Reggie* utilisant le protocole JRMP, *Java Remote Method Protocol*. C'est le protocole standard de RMI. En utilisant ce protocole à la place de la nouvelle couche JERI, on permet à des services ayant été conçus avec version antérieur à Jini 2.0 de s'enregistrer dans le service de recherche.

Le script ci-dessus fait appel à d'autres fichiers. Notamment, un fichier *policy.all* qui autorise *Reggie* à lire et écrire partout dans le système de fichiers, et un fichier de configuration *start-jrmp-reggie.config* qui fournit des informations utiles à *Reggie*, telle que l'adresse de téléchargement de son *proxy*.

En effet, *Reggie*, au même titre que les autres services met à disposition, sur un serveur web, son *proxy*.
Il ne faut donc pas oublier de démarrer un serveur web mettant à disposition les *proxies*.

8.1.1 Serveur web

Dans le cadre de ce projet, il a été choisi de séparer les *proxies* de services de celui du service de recherche.
Le *Starter kit* est livré avec une archive contenant des utilitaires. Parmi ceux-ci, se trouve un mini-serveur web pouvant servir à mettre des JAR à disposition.
Le script suivant permet de démarrer un serveur web sur la bibliothèque d'archives de base de Jini. Ce serveur est accessible sur le port 8081.

```
JINIHOME=/Users/mdespont/Documents/Ecole/teledomotique/jini2_0

java -jar $JINIHOME/lib/tools.jar \
     -port 8081 \
     -dir $JINIHOME/lib    &
```

Pour fonctionner correctement, *Reggie* doit être démarré sur une machine disposant d'un nom d'hôte qui soit accessible depuis le reste du réseau.

8.2 Démarrage du service

8.2.1 Serveur web

Avant de démarrer un service, il faut également démarrer un serveur web pour mettre à disposition son *proxy*. Comme pour le service de recherche, nous utilisons, le serveur web mis à disposition dans le JSK. Cette fois, le dossier accessible via le serveur web est le dossier `web` placé au même endroit que tous les dossiers de base des services. Ce serveur web est accessible sur le port 8082.

```
JINIHOME=/Users/mdespont/Documents/Ecole/teledomotique/jini2_0

java -jar $JINIHOME/lib/tools.jar \
     -port 8082 \
     -dir /Users/mdespont/Documents/Ecole/teledomotique/testjini/web &
```

8.2.2 Shell script

Pour démarrer le service *foodDistributor*, nous allons exécuter le script *start-foodDistributor.sh*

Ce script est semblable à celui utilisé pour démarrer *Reggie*. Toutefois, en plus, il spécifie le chemin d'accès aux archives *jini-core.jar* et *jini-ext.jar* qui sont dans la bibliothèque de base de jini. Ces archives sont nécessaires à toute application Jini.

Le fichier de configuration qui est passé en paramètre au service est *jeri-server.config*. C'est dans ce fichier que sont placées les informations relatives au service, notamment, sa localisation.

8.3 Démarrage du client

Le client est démarré à l'aide du script *start-foodDistributorClient.sh*. Nous avons ici un script très semblable à celui utilisé pour démarrer le serveur, à la différence que le client n'a pas de *proxy*, donc il n'y a pas besoin de spécifier son adresse.

8.4 En résumé

En résumé pour démarrer une fédération de service minimale, il faut démarrer :

1. le serveur web hébergeant le *proxy* du service de recherche ;
2. le serveur web hébergeant le *proxy* du service ;
3. le service de recherche (*reggie*) ;
4. le service ;
5. le client.

Ces opérations sont effectuée depuis le dossier de base du service, en exécutant les scripts suivants :

1. `mdespont% sh scripts/httpd-jini.sh`
2. `mdespont% sh scripts/httpd-service.sh`
3. `mdespont% sh scripts/start-reggie.sh`
4. `mdespont% sh scripts/start-foodDistributor.sh`
5. `mdespont% sh scripts/start-foodDistributorClient.sh`

Il est préférable de démarrer le service et le client dans des terminaux différents afin de pouvoir observer les messages qui s'affichent pendant l'exécution des applications.

9 Interface utilisateur graphique d'un service

9.1 Localisation de l'interface utilisateur

Jusqu'à présent, nous avons mis en place l'accès à un service. Une application cliente est capable d'utiliser les méthodes d'un service. Cependant, les appels de méthode à distance restent au niveau du programmeur. Pour qu'un utilisateur "standard" puisse utiliser le service, il faut lui faire une interface utilisateur.

Lorsque l'on parle interface homme-machine, le plus souvent, c'est à l'interface graphique que l'on pense, au GUI *Graphical User Interface*.
Malgré qu'il existe d'autres moyens pour établir une communication entre un homme et une machine, c'est tout de même le moyen le plus courant et le plus efficace que l'on connaisse.

Habituellement, une application est toujours très liée avec son interface utilisateur. Il n'est souvent pas possible de les dissocier.
Lorsque l'on dispose d'un modèle de programmation orienté service, il est plus facile de séparer l'interface utilisateur du service en lui même. Il est donc possible avec un tel modèle d'avoir plusieurs interfaces utilisateurs qui pilotent le même service.

Cependant, dans un tel modèle, où placer le code de l'interface utilisateur ?
Dans l'utilisation d'un service Jini, il est possible de mettre le code de l'interface graphique, dans le client du service ou dans le *proxy*.

Dans le cas où l'on place l'interface utilisateur dans l'application cliente, le service ne contrôle plus le client. Si le service change, le client n'est plus compatible, il faut le changer.

Dans le cas où c'est dans le *proxy* que l'on place le code, le service force le client a utiliser l'interface utilisateur qu'il a choisi. Il ne devient ainsi plus possible pour un utilisateur non humain, un programme, de se servir du service.

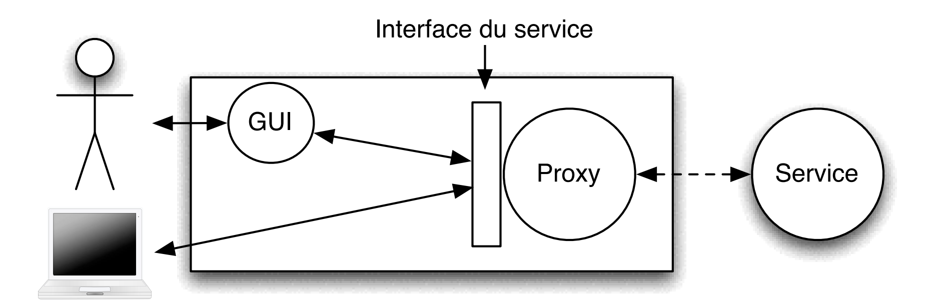


FIG. 7 – Un humain ou un ordinateur n'utilisent pas le service de la même manière

9.2 ServiceUI

Afin de résoudre le problème de la localisation du code de l'interface utilisateur, le projet *ServiceUI* [12] propose une solution.

Au lieu de placer l'interface utilisateur dans le client ou directement dans le *proxy*, on la place dans les attributs du service.

Cette technique permet à un programme d'utiliser le service directement via son interface. Mais elle permet également au service de proposer au client une interface utilisateur qu'il maîtrise.
Cette technique permet de laisser ouvertes toutes les possibilités d'interface utilisateurs.

Les spécifications du projet *ServiceUI* décrivent un objet `UIDescriptor` permettant de reconstruire une interface utilisateur. Cet objet est placé dans le tableau des attributs du `ServiceItem`.

L'objet `UIDescriptor` comporte plusieurs objets :
– `factory`

– `role`
– `toolkit`
– `attributes`

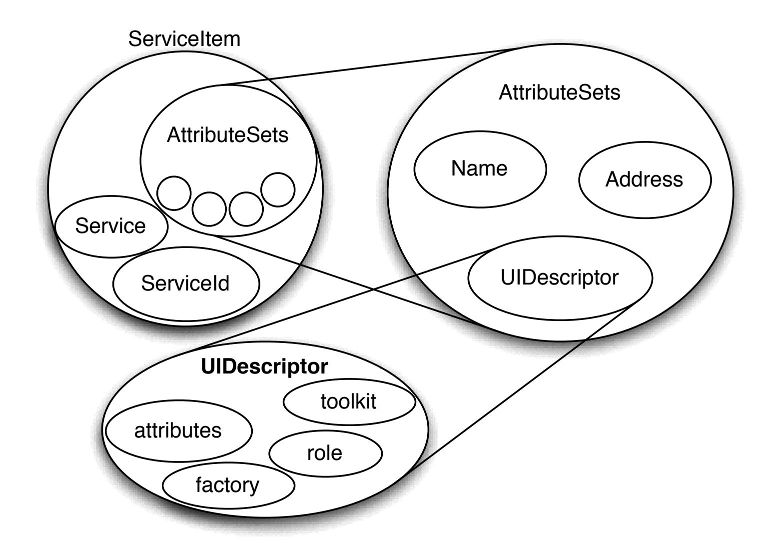


FIG. 8 – Composition de l'objet `UIDescriptor`

L'objet `factory` est une "fabrique" capable de reconstruire une interface graphique. Cet objet est de type `java.rmi.MarshalledObject`. En effet, l'objet `UIDescriptor` étant un attribut, de type `Entry`, il doit pouvoir implémenter l'interface `Serializable`.

L'objet `role` permet, par un chaîne de caractère, de décrire le rôle de l'interface utilisateur. En effet, on peut imaginer qu'un service mette à disposition une interface destinée à un utilisateur "standard" et une autre interface destinée à un administrateur du service. Le rôle permet de les distinguer.
Les spécifications du projet *ServiceUI* ne décrivent que trois rôles :
– MainUI.ROLE
– AdminUI.ROLE
– AboutUI.ROLE
L'objet `toolkit` permet d'indiquer par une chaîne de caractère le type de bibliothèque utilisée pour créer l'interface utilisateur. Cette information permet à un client qui serait limité pour une certaine interface, de pouvoir choisir une éventuelle version allégée de l'interface. Il est ainsi possible de fournir une interface graphique construite avec *Swing* pour les ordinateurs et une interface *AWT* pour les téléphones portables (encore que les téléphones portables capable d'utiliser des applications AWT ne sont pas légions).

Actuellement, le projet *ServiceUI* ne défini que des `toolkits` pour créer une interface utilisateurs graphique. Cependant, on peut imaginer définir un `toolkit` pour construire une interface utilisateur par synthèse et reconnaissance vocale, ou une interface graphique 3D immersive.
Un utilisateur d'applications domotiques pourrait ainsi avoir devant lui ou dans ses mains l'appareil qu'il manipule à distance.

L'objet `attributs` est un conteneur d'attributs pour l'interface utilisateur. Il est possible d'attribuer toutes sortes d'objets à l'interface utilisateur, pour autant qu'il puisse implémenter l'interface `Serializable`.

Dans le cadre de ce projet, plusieurs services de test on été réalisés, un service de distribution de nourriture pour les poissons, un service servant à allumer ou éteindre des lampes, et un service de webCam. Chacun de ses services à été doté d'une interface graphique conforme aux spécifications du projet *ServiceUI*.

Chaque service propose dans ses attributs une fenêtre de rôle MainUI.ROLE. Cette fenêtre propose à l'utilisateur tous les boutons, champs textes et autres composants utiles au pilotage du service.

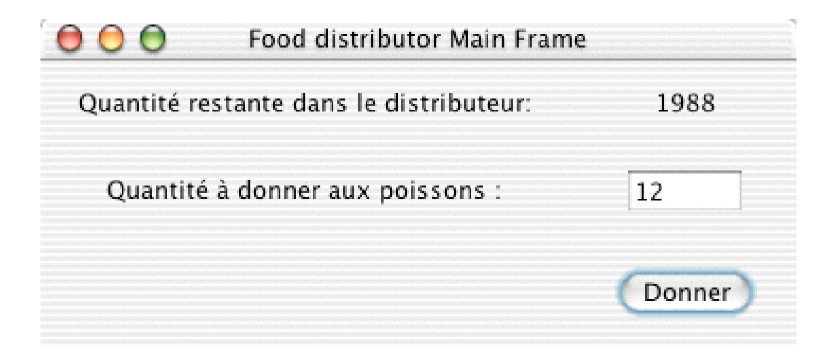


FIG. 9 – Interface utilisateur du distributeur de nourriture

Par cette méthode, le service peut proposer une interface graphique qu'il maîtrise et qui est de part ce fait la plus conforme avec l'utilisation que l'on peut faire du service. De plus, si un jour le le service est mis à jour, l'interface peut également être mise à jour sans que le client n'ait à être modifié.

10 Navigateur Jini

10.1 Principe du navigateur

Jusqu' à présent, nous avons toujours parlé du service et du client qui est capable de l'utiliser. Dans un service des plus simple, le client doit généralement connaître à l'avance les méthodes qu'il peut appeler sur le service. Le service et le client sont donc très liées.

Avec l'utilisation d'une interface graphique conforme aux spécifications du projet *ServiceUI*, on passe à l'étape suivante :
Un service à priori inconnu, est capable de fournir à un utilisateur humain, son service et de quoi l'utiliser. Et ceci, tout en garantissant que les programmes n'utilisant pas l'interface graphique puissent toujours fonctionner sans modification.

Malgré le fait qu'un service à priori inconnu puisse être utilisé, il subsiste toujours un problème. l'application cliente reste toujours liée au service qu'elle utilise. L'étape suivante consiste donc à permettre à un client d'utiliser n'importe quel service.

Un telle application est ce que l'on appelle un navigateur.

Un navigateur web permet d'afficher n'importe quelle page HTML. Un navigateur Jini permet d'afficher n'importe quelle interface utilisateur d'un service.

Dans le cas d'un navigateur web, on choisi la page à afficher en fournissant l'adresse de cette page au navigateur. Et si l'on ne connaît pas l'adresse, on fourni des critères à un moteur de recherche. Ce dernier retourne à l'utilisateur une liste des adresses qui correspondent.

Dans le cas d'un navigateur Jini, il est possible de fournir des critères à un service de recherche. Ce dernier va retourner à l'utilisateur des objets `ServiceItem` contenant les services qui correspondent aux critères de recherche. De plus, un `ServiceItem` peut contenir tout le code nécessaire à la construction de l'interface utilisateur.

10.2 Services contextuels, services connus

Un navigateur jini effectue la recherche des services :
- Soit dans les services de recherche visibles par diffusion *multicast*.
- Soit dans un service de recherche dont l'adresse est connue de l'utilisateur.

La première technique permet d'obtenir des services dit *contextuels*, qui sont les services à portée de diffusion *multicast*, donc physiquement proche. Il est ainsi possible à un utilisateur de "voir" des services dont il ne connaît pas forcément l'existence, mais qui de part leur proximité peuvent être utiles.
Il n'est pas utile d'utiliser sa machine à café depuis l'autre bout du monde, mais il peut être utile de le faire depuis son lit le matin pour se motiver à se lever !

La généralisation de services contextuels pour piloter tout les appareils d'une maison permettrait de fournir avec chaque appareil une interface graphique spatieuse et conviviale, choses qui ne sont pas toujours possible avec un mini-écran à cristaux liquide. Cela permettrait d'utiliser un appareil de taille raisonnable, genre PDA, pour afficher toutes les interfaces utilisateur de tous les appareils, et de continuer la miniaturisation des appareils et services.

La seconde technique de recherche utilise un service de recherche connu. C'est typiquement la situation, où un utilisateur désir enclencher le chauffage dans son chalet à la montagne.
Le chalet dispose d'un service de recherche dans lequel sont enregistrés tous les services du chalet. L'utilisateur va donc chercher uniquement le service chauffage dans le chalet. Le service de recherche du chalet est probablement hors de portée d'une diffusion *mulicast*. La seule technique applicable est donc d'utiliser l'adresse du service de recherche.

10.3 Critères de recherche avant ou après ?

Pour utiliser un service à l'aide d'un navigateur Jini, il faut pouvoir sélectionner le bon service. Il y a pour cela deux choix :
- Rechercher tous les services correspondant aux critères de recherche fournis.
- Obtenir tous les services, afficher leurs caractéristiques et laisser l'utilisateur choisir.

La première technique est la plus performante pour réduire au maximum le nombre de données échangées.
Toutefois, le nombre de critères possibles étant assez grand. Et les critères pas forcément les mêmes pour chaque service, l'introduction des critères de recherche est problématique. Pour faire une interface graphique, on peut utiliser se limiter à proposer un certain nombre de champs qui sont fréquemment remplis dans les services, tel que les champs adresse et localisation.

Après l'avoir testée, cette technique c'est révélée peu efficace. En effet, souvent les critères sont fournis de manière trop précise, et le service de recherche exclu les services qui pourraient correspondre, mais donc les attributs on été remplis de manière insuffisante.

Le navigateur qui a été conçu durant ce projet utilise donc l'autre possibilité : Il télécharge tout les services et laisse l'utilisateur choisir en fonction des informations présentes dans les attributs.
Les informations présentes ne sont pas forcément les mêmes pour chaque service. Cependant, il est plus facile de choisir une fois tous les services présents.

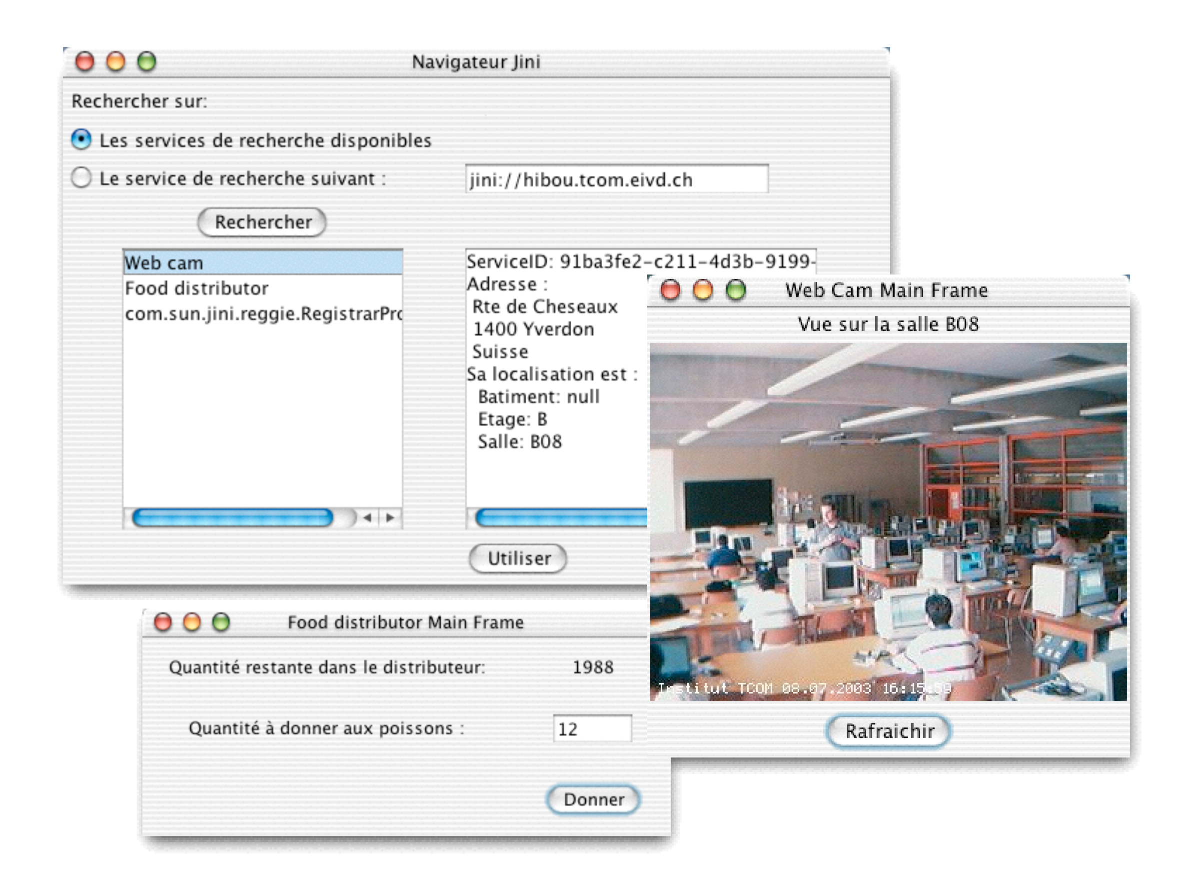


FIG. 10 – Le navigateur ainsi que la fenêtre de la WebCam et du distributeur de nourriture

Il reste une limitation au navigateur, il n'est pas capable d'afficher les champs dont il ne connaît pas le type. Ceci est un choix de programmation et non une véritable limitation.

Télécharger tous les services est possible seulement si le nombre de service n'est pas trop important. Avec le nombre restreint d'applications de domotique que l'on trouve dans les maisons actuellement, cette technique fonctionne parfaitement. Cependant, ce ne sera pas forcément le cas partout et pour toujours.
Un autre argument en faveur de cette technique, est le fait qu'un utilisateur qui décide d'utiliser un service particulier profitera certainement de l'occasion pour en utiliser d'autres. Il n'est donc pas totalement inutile de télécharger dans le navigateur tous les services.

10.4 Utilisation du navigateur

Le navigateur conçu dans ce projet est placé dans la même structure de dossier que les services. Il est compilé et démarré selon le même principe que les services et clients. Il dispose de son propre script Ant et de son propre *shell script*.

L'utilisation du navigateur est très simple. L'utilisateur qui désire employer un service effectue les opérations suivantes :

1. Démarrer le navigateur à l'aide de son script de démarrage.
2. Choisir s'il veut utiliser un service de recherche connu ou effectuer la recherche automatique.
3. Choisir le service voulu en fonction des attributs affichés dans le cadre de droite.
4. Sélectionner le service désiré dans la liste. Dans celle-ci, c'est le nom du service qui est affiché s'il est trouvé dans les attributs. Le cas échéant c'est le nom de la classe qui est affiché.
5. Cliquer sur le bouton utiliser. Le navigateur lance alors l'interface utilisateur du service.

L'interface utilisateur du service est reconstruite dans le navigateur. Chaque nouvelle fenêtre ainsi créée fait partie intégrante de cette application. La fermeture du navigateur ferme toutes les fenêtres lancées.
Dans l'autre sens, si le fait de cliquer sur la case de fermeture de la fenêtre d'un service quitte l'application, c'est tout le navigateur et les autres services qui sont quittés. Il faut bien faire attention à ce genre de détails au moment de la conception de l'interface utilisateur d'un service.

11 Sécurité

11.1 Motivations

Jusqu' à présent, nous nous sommes préoccupés que des fonctionnalités de l'infrastructure de télédomotique. Maintenant, nous allons nous préoccuper de la sécurité de ces fonctionnalités.
Pourquoi parler de sécurité ?

Et bien, par ce que l'architecture est conçue d'une telle manière qu'elle peut être vulnérable à des malveillances, ou qu'elle peut véhiculer des malveillances.

Il y a plusieurs problèmes :
– Du code malicieux peut être exécuté chez le client à son insu.
– N'importe quel utilisateur à le droit d'utiliser n'importe quel service.
– L'intégrité des données entre un client et un service n'est pas garantie.

L'architecture Jini étant basée sur le principe du code mobile, un client est souvent amené à exécuter du code qui vient d'ailleurs. Dans une utilisation habituelle, le client fait confiance au service et exécute le *proxy* de celui-ci sans se poser de questions.
Il suffit qu'une personne malveillante place un virus à la place du *proxy* et le client est infecté.

Autre faiblesse : un navigateur Jini démarré sur un réseau est capable de ”voir” tous les services de recherche disponibles, donc tous les services. Il suffit alors, pour n'importe quel utilisateur, de disposer d'un navigateur pour utiliser n'importe quel service.
Pourtant, il y a peut être des services qui sont critiques et que tout le monde ne doit pas être capable d'utiliser.

Dans le cas d'applications de domotique, personne n'a envie que des inconnus s'amusent avec les lampes de sa maison et déclenchent le chauffage, en plein hiver, et au milieu de la nuit !
Il est donc utile de pouvoir identifier l'utilisateur d'un service.

Depuis sa nouvelle version 2.0, Jini est capable d'intégrer les mécanismes de sécurités, SSL et Kerberos.

11.2 Principe de fonctionnement

Le principe fondamental du fonctionnement des mécanismes de sécurité est basé sur l'application de contraintes sur un *proxy*.

Une contrainte peut être appliqué par un client ou par le service. Dans les contraintes souvent utilisées, il y a :
- `Integrity`
- `ServerAuthentication`
- `clientAuthentication`

Lorsque l'on parle d'appliquer une contrainte, en fait on crée, à partir du *proxy* original, une copie qui est passée par plusieurs tests.

La contrainte `Integrity` oblige le client qui reçoit un *proxy* à vérifier l'intégrité des données qu'il reçoit. Le téléchargement du *proxy* s'effectue par un type de HTTP particulier. Le protocole *httpmd*.
Ce protocole utilise HTTP et lui ajoute une vérification de l'intégrité des données transmises, grâce à un *digest* MD5.
Une URL httpmd ressemble donc à ceci :
`http://monserveur/moncode.jar;md5=124367328748745874539845 3`

Le client qui reçoit une archive par httpmd crée de son coté un *digest* MD5 et le compare à celui contenu dans l'URL. Si les *digest* son égaux. L'archive a été téléchargée sans être modifiée.

Les contraintes `ServerAuthentication` et `clientAuthentication` obligent soit le service, soit le client, ou les deux à être authentifiés.
Couramment, on utilise SSL ou Kerberos pour faire des authentifications. Cela signifie que l'authentification va se faire par un mécanisme de clé privée, clé publique. Chacune des parties doit donc disposer de sa clé privée et de le clé public de l'autre. La gestion des clés est faite via des certificats X.509.

11.3 Droit d'accès au système de fichiers

Un autre niveau de sécurité, est celui des droits d'accès de l'application dans le système de fichiers.
Il est possible pour toute application Java d'indiquer exactement quelle méthode à le droit d'accéder à quelle partie du système de fichier.
Habituellement cette opération est effectuée via un fichier *policy*. Cependant, avec ce mécanisme, l'application doit lire le fichier de configuration avant d'être démarrée. Ce qui nous empêche d'utiliser ce mécanisme pour attribuer des droits à un *proxy* qui est un morceau d'application arrivant en cours d'exécution.

Un nouveau mécanisme à donc été utilisé. Pour utiliser ce mécanisme d'attribution dynamique des droits d'accès, une application doit spécifier, avant son démarrage l'utilisation d'un autre fournisseur de droit d'accès.
On utilise : `policy.provider=net.jini.security.policy.DynamicPolicyProvider`

Ce fournisseur de droits d'accès se trouve dans l'archive jsk-policy.jar qui doit être installé dans le dossier d'extension de Java. (voir : installation de Jini dans l'annexe A)

11.4 Etapes pour sécuriser un *proxy*

Les différentes étapes nécessaires à un client pour utiliser un *proxy* "sécurisé" sont les suivantes :

1. Le bout client est extrait du *proxy* téléchargé de manière non sécurisée.
2. Le bout client est utilisé pour contacter le service et lui demander de s'authentifier.
3. Si l'authentification est acceptée, le serveur retourne un objet `verifier`.
4. Le `verifier` permet au service de vérifier lui aussi le *proxy*.
5. Les droits d'accès sont donnés de manière dynamique au *proxy*.
6. Le *proxy* est autorisé à effectuer des appels de méthodes à distance avec le serveur.

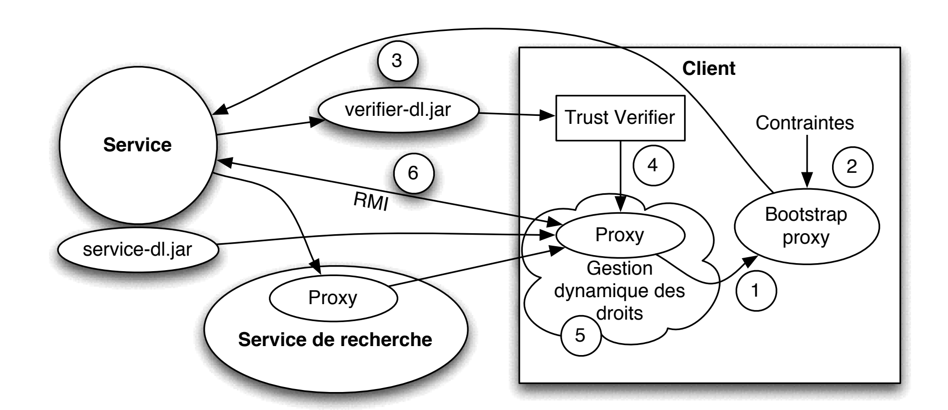


FIG. 11 – Les différentes étapes pour vérifier un *proxy*

11.5 Certificat X.509

Précédemment, nous avons parlé de clé publique et clé privée. Afin de gérer de manière cohérente toutes ces clés. Nous utilisons des certificats. Le certificat est un conteneur qui est capable de stocker dans une même entité, des clés et des informations sur le détenteur de la clé. Un certificat à une durée de validité et peut être signé par une autorité de certification.

Dans notre cas, un certificat va être fourni à chaque service ou client. Le service de recherche sera également certifié.
Un certificat est en général présenté sous la forme d'un fichier .cert. Dans notre cas, on va encore inclure le certificat dans un *porte-clé* : le Keystore. C'est un format qui est propriétaire à SUN. Chaque service va donc posséder son propre porte clé service.keystore pour garder sa clé privée.
De plus, chaque service aura aussi à disposition dans un lieu publique, un serveur web par exemple, un porte clé contenant les clé publique de tous les services et clients avec qui il doit communiquer. ce porte clé s'appelle truststore.

Les certificats sont créés avec l'utilitaire keytool fourni avec java.

```
keytool -genkey -dname "cn=Food Distributor, ou=Tcom, o=eivd, c=CH"
      -alias foodDistributor -keypass password
       -keystore toto/server.keystore
      -storepass keystorepwd -validity 180
```

Cette commande permet de générer, pour 180 jours, pour le service *foodDistributor*, une paire de clé et de garder la clé privée dans un porte-clé nommé server.keystore.

11.6 Fichiers de configuration

Les fichiers de configuration que nous avions précédemment sont complétés par de multiples instances d'objets permettant d'authentifier des services et d'effectuer des communications sécurisées.
Un fichier de configuration supplémentaire est présent, il s'agit d'un fichier .login qui permet de définir le type de contrôle d'authentification JAAS, *Java Authentication and Autorisation Service* à utiliser et dans quel porte clé aller chercher les clés.
Il est possible en configurant le JAAS d'afficher un boîte de dialogue qui demande le mot de passe pour accéder au porte clé, ou encore d'utiliser directement des mots de passe se trouvant dans des fichiers.

12 Installation de l'infrastructure de télédomotique

12.1 Plate-forme supportées

A priori, toutes les plate-formes disposant d'un machine virtuelle Java (JDK-1.4) sont capables d'exécuter un service ou un navigateur Jini.

Cependant, face à la complexité des commandes à écrire pour démarrer une application, des *shell scripts* ont été écrits. Ceux-ci ne s'exécutent que sur des machines de type Unix.
Il n'est toutefois pas compliqué de modifier ces scripts pour en faire des *.bat* s'exécutant sur un système Windows.

Les systèmes qui ont été testés, sont Mac OS X et Linux Mandrake.

12.2 Bibliothèques requises

Les différentes applications composant l'infrastructure ont été réalisées à l'aide de plusieurs bibliothèques :
– la bibliothèque de base de java ;
– la bibliothèque de Jini 2.0 ;
– la bibliothèque de *ServiceUI* pour les services avec interface utilisateur.

Il est donc nécessaire d'avoir installé ces bibliothèques avant de vouloir utiliser les applications développées dans ce projet.
Les manuels d'installation de Jini 2.0 et *ServiceUI* sont disponibles dans les annexes A et B.

12.3 Installation

Nous allons procéder à l'installation des services livrés en exemples, du navigateur, et de tous les scripts associés.

1. Copier le dossier de base de tout les services, ici appelé *testjini*, à l'endroit voulu.

2. Indiquer le chemin d'accès au dossier Jini2.0 dans la variable jini.home du fichier de configuration de la compilation, build.properties. Ceci doit être fait pour tous les fichiers build.properties. Il y en a un par application, cependant à l'installation ces fichiers sont tous identiques.
3. Indiquer le chemin d'accès au dossier Jini2.0 dans la variable JINIHOME se trouvant dans tous les *shell scripts*
4. S'assurer que la machine hôte du service de recherche dispose d'un nom d'hôte atteignable depuis les services et les clients.

12.4 Compilation

La compilation des applications de l'infrastructure a été réalisée, dans le cadre de ce projet, à l'aide de l'outils Ant. Bien que l'on puisse effectuer cette compilation avec d'autres outils, il est recommandé d'utiliser Ant.
Le manuel d'installation de Ant est disponible en annexe C.

13 Création de nouveaux services

Les services déjà présents en exemple sont très proches de tous les services que l'on pourrait ajouter. Nous avons les services *foodDistributor* et *lightControler* qui sont des services dont le *proxy* utilise RMI pour contacter le service.
Et nous avons le service *WebCam* qui utilise le *proxy* uniquement pour transmettre au client l'adresse de l'image de la webcam.
Nous avons là de quoi réaliser déjà quelques types de service différents.

Il est donc assez aisé de refaire un service à l'aide des exemples fournis.
Les étapes sont les suivantes :

1. Définir l'interface que l'on désire.
2. Reprendre le code source d'un service existant, remplacez l'implémentation du service existant par celle du nouveau service.
3. Remplacer dans tout les fichiers relatif au service (sources, *shell scripts*, fichiers Ant et fichiers de configuration) le nom de l'ancienne interface par le nom de la nouvelle.
   Par exemple : remplacer *foodDistributor* par *coffeeMachine*.
4. Modifier dans les fichiers de configuration, les informations relatives au service, comme son nom, ou sa localisation physique.
5. Remplacer, dans les bons fichiers, le code construisant l'ancienne interface utilisateur par le code d'une interface graphique qui soit capable d'utiliser le service.
6. Compiler le service.
7. Voilà, il ne reste plus qu'à le démarrer et l'utiliser.

Un service comme celui là est vite fait, mais il ne fait rien !

En effet, le code du service en lui même reste à écrire. Ce qui est fourni ici, c'est juste un moyen d'accès.
Pour piloter une machine à café, il reste encore du chemin.

Mais il ne faut pas se démoraliser, c'est possible !

14 Améliorations possibles

Une telle infrastructure, et tout logiciel de manière générale, n'est jamais fini. Il y a toujours des améliorations possibles.

La première de ces améliorations sera d'augmenter le nombre de possibilités d'accès à la fédération de service d'une maison.

Jini est une architecture souple au niveau des types de réseaux utilisés, pourtant, dans bien des cas elle ne l'est pas assez. De plus, Jini est tout de même une architecture assez lourde pour des machines qui sont limitées en ressources.

Pour tenter de contourner certains problèmes, il est utile de disposer d'un plus large éventail de possibilités permettant d'accéder à la fédération de service.
Parmi ces possibilités, il y a :

- La *Servlet*/client Jini. C'est un client Jini capable d'afficher une interface utilisateur web.
- Une application RMI relais pour une machine n'ayant pas assez de ressources pour utiliser l'architecture Jini, mais qui en a suffisamment pour utiliser RMI.

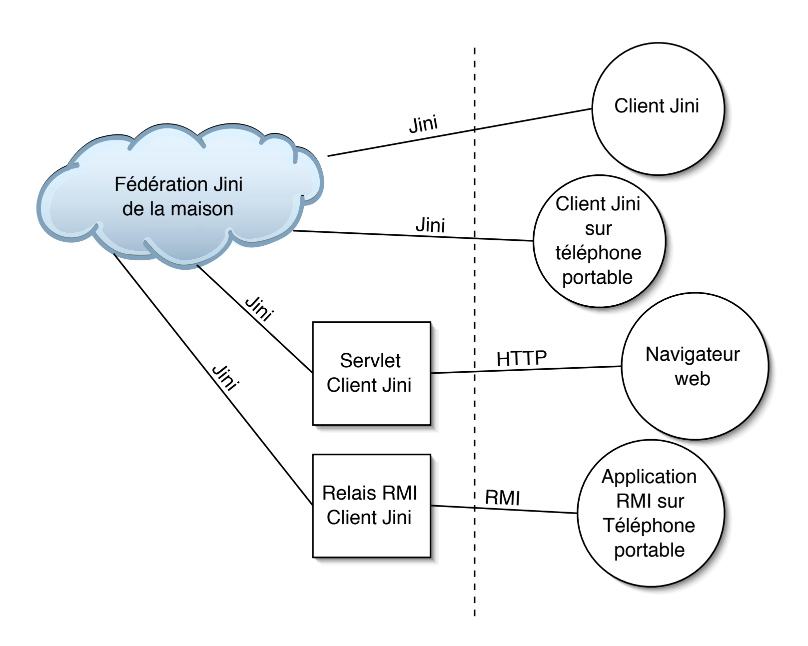


FIG. 12 – De multiples passerelles permettent l'accès à la fédération jini

Une autre amélioration possible est la gestion des langues.
Ceci peut être aisément réalisé en plaçant le nécessaire à la gestion des langues dans les attributs d'un service. Chaque service est conçu avec une interface utilisateur par langue. Celle-ci sont disponibles dans l'objet `UIDescriptor` qui est un attribut.
Le navigateur, au moment de choisir l'interface utilisateur qu'il désire employer, choisi celle qui est dans sa langue.

Une amélioration qui vaudrait la peine d'être réalisée rapidement, est la réalisation d'un installeur convivial. Cela permettrait à n'importe qui d'utiliser l'infrastructure.

Dans le même ordre d'idée, une application capable de gérer les services, serait la bienvenue. Il serait ainsi possible de se libérer de l'utilisation des *shell scripts* qui peuvent paraître rebutants pour un utilisateur ”standard”.

Il serait intéressant d'utiliser le service *mercury* qui est un boîte au lettres d'événements. Les services pourraient ainsi réagir à des événements, et communiquer, entre eux, sans intervention humaine suivant les événement survenus.

15 Conclusions

Une infrastructure de télédomotique n'est pas chose des plus facile a réaliser. En effet, lorsque l'on touche à la domotique, on est vite confronté au problème de la diversité énorme du matériel, et aux ressources mémoire et processeurs souvent très limitées des appareils.

L'architecture Jini semble être assez souple pour remplir la difficile tâche de fédérer toutes sortes de service. Et ceci, indépendamment des moyens de communication employés et des machines hôtes.

Cependant, il faut parfois modifier un peu l'architecture ”normale” pour déplacer les fonctions un peu trop gourmandes pour un service sur une machine hôte qui dispose de plus de ressources. Cette gymnastique est parfois un peu compliquée, mais au final, l'utilisateur *n'y voit que du feu.* Ce qui est le but d'une infrastructure de télédomotique.

Jini est une architecture souple, car ces concepteurs ont toujours eu le souci de généraliser au maximum les concepts. Ils sont, parfois même, trop généralisés, ce qui entrave quelquefois la compréhension. Il est parfois nécessaire de construire plusieurs objets pour en obtenir un qui permet d'en obtenir un autre. Ces enchaînements sont souvent des plus difficiles à comprendre. Seul un bon exemple permet parfois d'avancer.

Toutefois, ces exemples sont parfois difficiles à trouver. En effet, l'architecture Jini a subi de profondes modifications, en passant à la version 2.0, en Juin 2003. Et la plupart des exemples qu'il est possible de trouver sont encore à des versions antérieures. Certaines nouveautés introduites avec la version 2.0, comme la sécurité par exemple, n'ont pas encore eu vraiment le temps d'être éprouvées par une large communauté de programmeurs. Pour preuve, je n'ai trouvé aucun exemple de code utilisant la sécurité, hormis l'exemple fourni avec le *Starter kit*.
Seule la *mailing-list* des utilisateurs de Jini permet de disposer d'une source d'information récente.

Il semble tout de même que l'arrivée de la version 2.0 de Jini a donné un coup de fouet au programmeurs Jini. Fréquemment, de nouveaux projets basés sur Jini voient le jour.

La percée, ces derniers temps, des accès internet haut-débit et des réseaux domestique sans fil n'a cessé de s'accroître. Cette combinaison de réseau permet à un particulier d'installer, pour un prix à sa portée, un réseau dans toute une maison d'un seul coup.
Cela offre de grandes perspectives à la télédomotique...

Mathieu Despont

Références

[1] JATON M.
Infrastructure de télédomotique : Cahier des charges
Institut TCOM, 2003

[2] *Universal Plug and Play Forum*
http://www.upnp.org

[3] *Site officiel de Salutation*
http://www.salutation.org

[4] *Site officiel de la communauté Jini*
http://www.jini.org

[5] DESPONT M.
Infrastructure de télédomotique : Projet de semestre
Institut TCOM, projet de semestre 2003

[6] DESPONT M.
Jini
Institut TCOM, mini-présentation 2003

[7] *Site du projet Surrogate*
http://surrogate.jini.org

[8] *Site web du projet Davis.*
http://davis.jini.org

[9] *ANT : Un outils de compilation*
http://ant.apache.org

[10] *IDE pour développer et manager des services Jini*
http://www.incax.com

[11] *Homepage d' Alexander Shvets : Plusieurs exemples d'applications Jini compilées avec* Ant
http://home.earthlink.net/˜shvets/

[12] *Spécifications du projet ServiceUI qui tend à normaliser les GUI d'applications Jini.*
http://www.artima.com/jini/serviceui/

[13] *Jini Corner at Artima : Documentation Jini, notamment sur ServiceUI et sur la sécurité.*
http://www.artima.com/jini/

[14] *Outils pour débuter avec Jini 2.0*
http://startnow.jini.org/

[15] *Outils pour débuter avec Jini 2.0*
http://www.dancres.org/cottage/starting_jini.html

[16] *Projet français réalisé avec Jini*
http://www.lrde.epita.fr/˜ricou/

[17] *Turoriel de Jan Newmarch sur l'architecture Jini*
http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml

[18] *Archives de la mailing list des utilisateurs de Jini*
http://swjscmail1.java.sun.com/cgi-bin/wa?A0=jini-users

[19] *Le projet java X10 : Piloter des appareils X10 avec des programmes java.*
http://x10.homelinux.org/

[20] *Bus de terrain EIB : Très utilisé par les électriciens en Europe.*
http://www.eiba.com/home.nsf

[21] *Ecole d'ingénieurs du Canton de Vaud,*
http://www.eivd.ch

[22] *Institut Tcom de l'école d'ingénieur du canton de vaud,*
http://www.tcom.ch

A Installation de Jini 2.0

Attention, l'architecture Jini est soumise à la licence SCSL. Cette licence est un peu particulière, c'est une licence Open source, mais seulement pour ceux qui acceptent la licence !
Cette licence est conçue pour avoir les avantages d'une communauté open source qui vérifie et complète le code fourni. Et d'un autre coté, SUN se garde des droits, dont surtout celui de percevoir un peu d'argent sur les applications commerciales conçues sur la base de Jini. Les applications non commerciales étant toujours libres.

Voici les différentes étapes d'installation de la bibliothèque Jini 2.0 :

1. Télécharger le *Jini Starter Kit* sur le site de la communauté Jini : `www.jini.org`.
2. Décompressez le fichier jini-2 0-src.zip.
3. Placez le dossier jini2 0 à l'endroit de votre choix.
4. Extraire l'archive jsk-policy.jar se trouvant dans le dossier jini2 0/lib/ et la placer dans le dossier d'extension de java : j2sdk1.4/jre/lib/ext/ ou pour MacOS X : /System/Library/Java/Extensions/.

B Installation de *ServiceUI*

Le projet *ServiceUI* tend à normaliser la conception d'interface utilisateur pour les services Jini. Dans ce but, une petite bibliothèque a été écrite. Pour l'utiliser nous allons simplement placer l'unique archive JAR serviceui-1.1.jar dans le dossier bibliothèque de Jini : jini2 0/lib/.

Cette archive est disponible sur le site de *ServiceUI* : `http://www.artima.com/jini/serviceui/` ou sur le site de la communauté Jini dans son espace projet : `http://serviceui.jini.org`.

C Installation de Ant

Ant est l'outil qui est utilisé dans ce projet pour compiler les applications et créer des archives JAR.

Les différentes étapes pour l'installation sont les suivantes :

1. Ant peut être téléchargé à l'adresse suivant : `http://ant.apache.org/`.
2. Placer le dossier de Ant à l'endroit de votre choix.
3. Ajouter le dossier bin/ de Ant à votre variable d'environnement path.

4. Définir la variable d'environnement ANT HOME au dossier de base de Ant. temDéfinir la variable d'environnement JAVA HOME au dossier de base de Java

Voici un exemple des lignes de code possible pour effectuer ces tâches.

C.1 Installation sur Linux

Avec un interpréteur *bash* commun sous Linux :

```
export ANT_HOME=/usr/local/ant
export JAVA_HOME=/usr/local/jdk-1.4.0
export PATH=${PATH}:${ANT_HOME}/bin
```

C.2 Installation sur MacOS X

Avec un interpréteur *csh* commun sous MacOS X (Darwin) :

```
setenv ANT_HOME /usr/local/ant
setenv JAVA_HOME /System/Library/Frameworks/JavaVM.framework/Home
set path=( $path $ANT_HOME/bin )
```